图文 101 在初始化BrokerController的时候，都干了哪些事情?

详情 评论

在初始化BrokerController的时候，都干了哪些事情?

继《从零开始带你成为JVM实战高手》后，阿里资深技术专家携新作再度出山，重磅推荐：

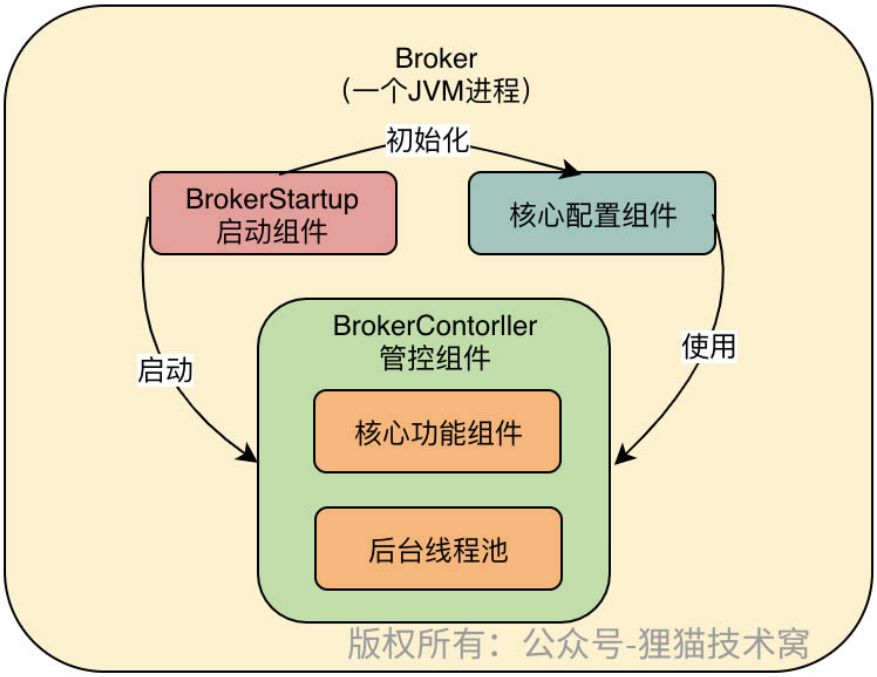
（点击下方蓝字试听）

《从零开始带你成为MySQL实战优化高手》

1、BrokerController创建完之后是在哪里初始化的?

接着上一讲，我们继续说，现在大家已经了解到了Broker作为一个JVM进程启动之后，是BrokerStartup这个启动组件，负责初始化核心配置组件，然后启动了BrokerController这个管控组件。然后在BrokerController管控组件中，包含了一大堆的核心功能组件和后台线程池组件。

现在我们来看一下下面的图，已经表达了上面的意思。



接着我们来看一下，那现在BrokerController都创建好了，里面的一大堆核心功能组件和后台线程池都创建好了，接下来他还要做一些初始化的工作，这个触发BrokerController初始化的代码在哪里呢？

其实还是在createBrokerController()方法里，在你创建完了BrokerController之后，就有一个初始化的代码，看下面的代码和注释。

```
// 我们能看到，在这里触发了BrokerController的初始化
boolean initResult = controller.initialize();
if (!initResult) {
    controller.shutdown();
    System.exit(-3);
}
// 这里就是注册了一个JVM的关闭钩子
// JVM退出的时候，其实就会执行里面的回调函数，
//本质也是释放一堆资源
Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
    private volatile boolean hasShutdown = false;
    private AtomicInteger shutdownTimes = new AtomicInteger(0);
    @Override
    public void run() {
        synchronized (this) {
            log.info(
                "Shutdown hook was invoked, {}",
                this.shutdownTimes.incrementAndGet());
            if (!this.hasShutdown) {
                this.hasShutdown = true;
                long beginTime = System.currentTimeMillis();
                controller.shutdown();
                long consumingTimeTotal =
                    System.currentTimeMillis() - beginTime;
                log.info(
                    "Shutdown hook over, consuming total time(ms): {}",
                    consumingTimeTotal);
            }
        }
    }
}, "ShutdownHook"));
// 最后这个createBrokerController()方法
// 就是返回创建和初始化好的BrokerController了
return controller;
```

2、一步一步分析BrokerController初始化的过程

接着我们一步一步的分析BrokerController初始化的过程，大家看下面的源码和注释就可以了，其实很多东西你现在看一下我写的注释有个了解就行了，真的不用过于的深究，有时候刚开始你深究过多了，就会导致你发现大脑一片混乱，最后就直接放弃看源码了，所以这里大致有一个BrokerController初始化的过程就行了。

下面就是BrokerController.initialize()方法的完整的源码分析，大家重点看注释。

```
// 首先开头4行代码，他其实就是在加载一些磁盘上的数据到内存
// 比如加载Topic的配置、Consumer的消费offset、
// Consumer订阅组、过滤器
// 这些东西如果都加载成功了，那么result必然是true
boolean result = this.topicConfigManager.load();
result = result && this.consumerOffsetManager.load();
result = result && this.subscriptionGroupManager.load();
result = result && this.consumerFilterManager.load();

// 看看如果一大堆数据加载成功了，这里会干什么
if (result) {
    try {
        // 首先这里创建了消息存储管理组件，
        //一看就是管理磁盘上的消息的
        this.messageStore =
            new DefaultMessageStore(
            this.messageStoreConfig,
            this.brokerStatsManager,
            this.messageArrivingListener,
            this.brokerConfig);

        // 这儿是如果启用了dleger技术进行主从同步以及管理commitlog
        // 他就要初始化一堆dleger相关的组件
        if (messageStoreConfig.isEnableDLegerCommitLog()) {

            DLedgerRoleChangeHandler roleChangeHandler =
                new DLedgerRoleChangeHandler(
                this,
                (DefaultMessageStore) messageStore);

```

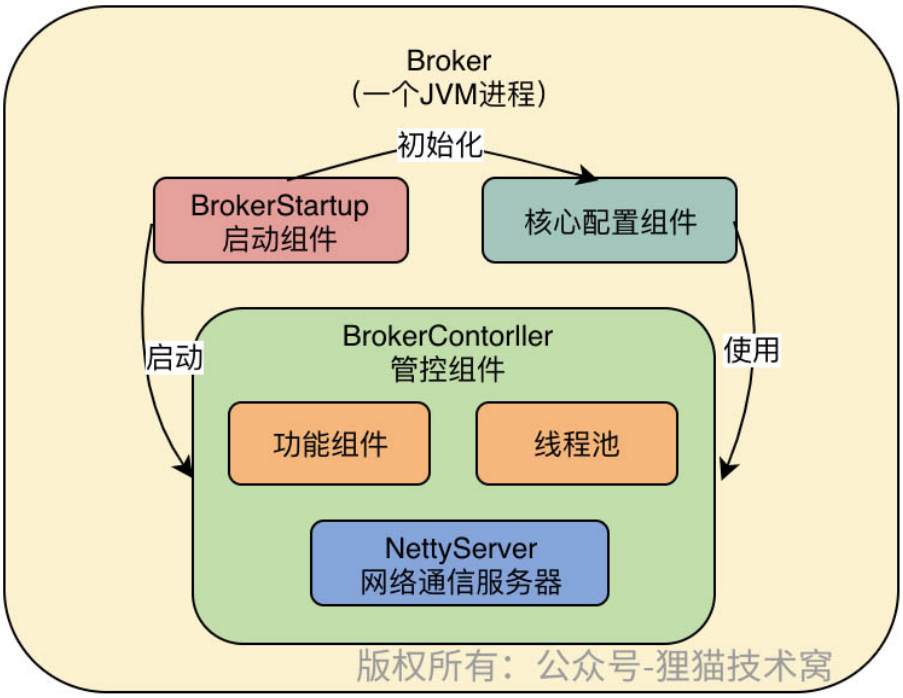
```
        ((DLedgerCommitLog)((DefaultMessageStore) messageStore).
                    getCommitLog()).
                    getdLedgerServer().
                    getdLedgerLeaderElector().
                    addRoleChangeHandler(roleChangeHandler);
        }
        // 这是Broker的统计组件
        this.brokerStats =
            new BrokerStats(
                (DefaultMessageStore) this.messageStore);

        // 下面的代码，现在暂时看不懂也没事
        MessageStorePluginContext context =
            new MessageStorePluginContext(
             messageStoreConfig,
             brokerStatsManager,
             messageArrivingListener,
             brokerConfig);

        this.messageStore =
            MessageStoreFactory.build(
                context, this.messageStore);

        this.messageStore.
            getDispatcherList().
            addFirst(
                new CommitLogDispatcherCalcBitMap(
                    this.brokerConfig,
                    this.consumerFilterManager));

    } catch (IOException e) {
        result = false;
        log.error("Failed to initialize", e);
    }
}
```

看完上面那一大坨代码，大家有什么感觉?

要我说，感觉就是没感觉，其实很多人平时自己看源码，看到这里就开始痛苦了，觉得真的看不懂，整个人陷入极大的挫败感和痛苦之中。其实完全没必要，上述代码你其实有一个印象就可以了，不用现在过于较真。

```
// 这个地方就比较关键了，我们看这里，
// Broker也要接受人家的请求的

// 比如Producer发送请求过来发消息，
// Consumer发送请求过来拉取消息

// 所以Broker必然也是有Netty服务器的
this.remotingServer =
    new NettyRemotingServer(
        this.nettyServerConfig,
        this.clientHousekeepingService);

NettyServerConfig fastConfig =
    (NettyServerConfig) this.nettyServerConfig.clone();

fastConfig.setListenPort(
    nettyServerConfig.getListenPort() - 2);

this.fastRemotingServer =
    new NettyRemotingServer(
        fastConfig,
        this.clientHousekeepingService);
```

讲到这里，我们先在下面的图里补充一下Netty服务器的概念，让大家看到，BrokerController里其实也会包含核心的Netty服务器，用来接收和处理Producer以及Consumer的请求。




```
// 从这里往下，一大堆的代码,在初始化一些线程池

// 这些线程池，有的是负责处理请求的线程池，
//有的是在后台运行的线程池

// 比如这个，sendMessageExecutor,
//一看就是人家发送消息过来的处理线程池

this.sendMessageExecutor =
    new BrokerFixedThreadPoolExecutor(

     this.brokerConfig.
        getSendMessageThreadPoolNums(),

     this.brokerConfig.
        getSendMessageThreadPoolNums(),

     1000 * 60,
     TimeUnit.MILLISECONDS,
     this.sendThreadPoolQueue,
     new ThreadFactoryImpl(
         "SendMessageThread_"));

```

```
// 这个一看就是处理consumer拉取消息的线程池
this.pullMessageExecutor =
    new BrokerFixedThreadPoolExecutor(

        this.brokerConfig.
            getPullMessageThreadPoolNums(),

        this.brokerConfig.
            getPullMessageThreadPoolNums(),

          1000 * 60,
          TimeUnit.MILLISECONDS,
          this.pullThreadPoolQueue,

          new ThreadFactoryImpl(
              "PullMessageThread_"));

// 这个一看就是回复消息的线程池
this.replyMessageExecutor =
    new BrokerFixedThreadPoolExecutor(

      this.brokerConfig.
            getProcessReplyMessageThreadPoolNums(),

      this.brokerConfig.
            getProcessReplyMessageThreadPoolNums(),
      1000 * 60,
      TimeUnit.MILLISECONDS,
      this.replyThreadPoolQueue,
      new ThreadFactoryImpl(
          "ProcessReplyMessageThread_"));

```

```
// 这个一看就是查询消息的线程池
this.queryMessageExecutor =
    new BrokerFixedThreadPoolExecutor(

      this.brokerConfig.
        getQueryMessageThreadPoolNums(),

      this.brokerConfig.
        getQueryMessageThreadPoolNums(),

      1000 * 60,
      TimeUnit.MILLISECONDS,
      this.queryThreadPoolQueue,

      new ThreadFactoryImpl(
          "QueryMessageThread_"));

// 这个一看就是管理broker的一些命令执行的线程池
this.adminBrokerExecutor =
   Executors.newFixedThreadPool(
    this.brokerConfig.getAdminBrokerThreadPoolNums(),
    new ThreadFactoryImpl("AdminBrokerThread_"));

// 这个就是管理客户端的线程池
this.clientManageExecutor = new ThreadPoolExecutor(

   this.brokerConfig.
    getClientManageThreadPoolNums(),

   this.brokerConfig.
    getClientManageThreadPoolNums(),

```

```
    1000 * 60,
    TimeUnit.MILLISECONDS,
    this.clientManagerThreadPoolQueue,

    new ThreadFactoryImpl(
        "ClientManageThread_"));

// 这个就比较关键了，一看就是个后台线程池，
// 负责给nameserver发送心跳的
this.heartbeatExecutor =
     new BrokerFixedThreadPoolExecutor(

         this.brokerConfig.
             getHeartbeatThreadPoolNums(),

         this.brokerConfig.
             getHeartbeatThreadPoolNums(),

         1000 * 60,
         TimeUnit.MILLISECONDS,
         this.heartbeatThreadPoolQueue,

         new ThreadFactoryImpl(
             "HeartbeatThread_", true));

// 这个是结束事务的线程池，这个一看就是跟事务消息有关系的
this.endTransactionExecutor =
     new BrokerFixedThreadPoolExecutor(

         this.brokerConfig.
             getEndTransactionThreadPoolNums(),

         this.brokerConfig.
             getEndTransactionThreadPoolNums(),

         1000 * 60,
         TimeUnit.MILLISECONDS,
         this.endTransactionThreadPoolQueue,

         new ThreadFactoryImpl(
             "EndTransactionThread_"));

// 这个一看就是管理consumer的线程池
this.consumerManageExecutor =
    Executors.newFixedThreadPool(
       this.brokerConfig.
             getConsumerManageThreadPoolNums(),

       new ThreadFactoryImpl("ConsumerManageThread_"));

this.registerProcessor();

// 下面这些代码，一看就是开始定时调度一些后台线程执行了
final long initialDelay =
     UtilAll.computeNextMorningTimeMillis() -
         System.currentTimeMillis();

final long period = 1000 * 60 * 60 * 24;

// 下面就是定时进行broker统计的任务

this.scheduledExecutorService.
     scheduleAtFixedRate(new Runnable() {

```

```
153     @Override
154     public void run() {
155         try {
156             BrokerController.this.
157                 getBrokerStats().
158                 record();
159         } catch (Throwable e) {
160             log.error("schedule record error.", e);
161         }
162     }
163 }, initialDelay, period, TimeUnit.MILLISECONDS);
164 
165 // 下面就是定时进行consumer消费
166 // offset持久化到磁盘的任务
167 this.scheduledExecutorService.scheduleAtFixedRate(
168     new Runnable() {
169         @Override
170         public void run() {
171             BrokerController.this.
172                 consumerOffsetManager.persist();
173         }
174       },
175     1000 * 10,
176     this.brokerConfig.getFlushConsumerOffsetInterval(),
177     TimeUnit.MILLISECONDS);
178 
```

```
179 // 下面就是定时对consumer filter过滤器进行持久化的任务
180 this.scheduledExecutorService.scheduleAtFixedRate(
181     new Runnable() {
182         @Override
183         public void run() {
184            BrokerController.this.
185                consumerFilterManager.
186                persist();
187         }
188     },
189     1000 * 10,
190     1000 * 10,
191     TimeUnit.MILLISECONDS);
192 
193 // 下面是定时进行broker保护的任务
194 this.scheduledExecutorService.scheduleAtFixedRate(
195     new Runnable() {
196         @Override
197         public void run() {
198            BrokerController.this.protectBroker();
199         }
200     },
201     3,
202     3,
203     TimeUnit.MINUTES);
204 
```

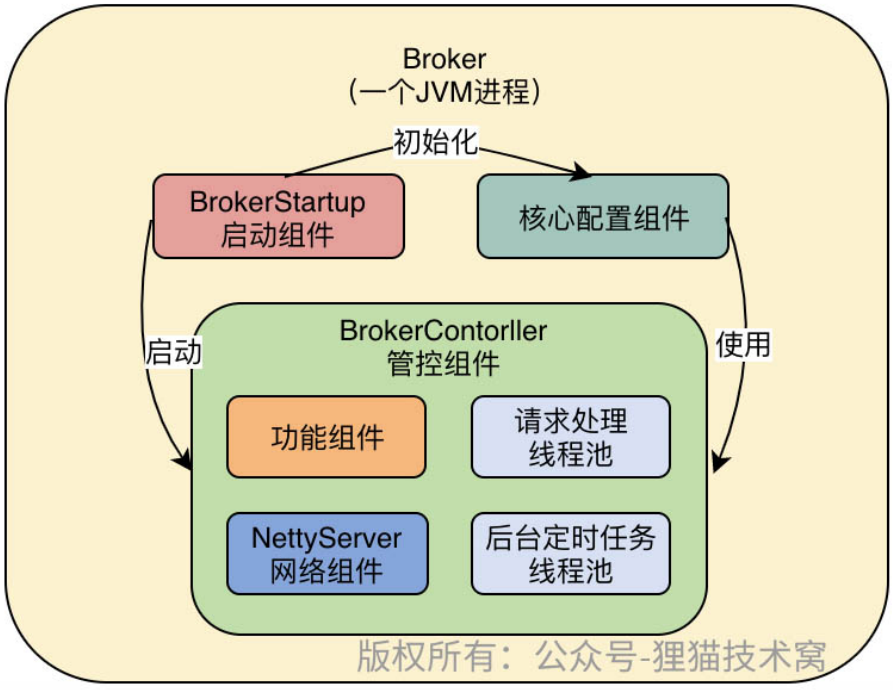
```
// 下面是定时打印watermark，就是水位的任务
this.scheduledExecutorService.scheduleAtFixedRate(
    new Runnable() {
        @Override
        public void run() {
           BrokerController.this.printWaterMark();
        }
    },
    10,
    1,
    TimeUnit.SECONDS);

// 下面是定时进行落后commitlog分发的任务
this.scheduledExecutorService.scheduleAtFixedRate(
    new Runnable() {
        @Override
        public void run() {
            log.info(
              "dispatch behind commit log {} bytes",

              BrokerController.this.
                 getMessageStore().
                 dispatchBehindBytes());
         }
    },
    1000 * 10,
    1000 * 60,
    TimeUnit.MILLISECONDS);
```

估计很多人看完了上面一大堆的处理请求的线程池的初始化和启动后台定时调度任务的代码，都一脸的懵逼，但是稍微找到点感觉了。

毕竟每个人都知道，后续Broker要处理一大堆的各种请求，那么不同的请求是不是要用不同的线程池里的线程来处理？

然后Broker要执行一大堆的后台定时调度执行的任务，这些后台定时任务是不是要通过线程池来调度定时任务？

所以其实你只要理解到这个程度就可以了，所以此时我们对下面的图又做了一些改动，在里面引入了两种线程池的概念，一种线程池是用来处理别人发送过来的请求的，一种线程池是执行后台定时调度任务的。



我们接着往后看剩余的代码，很多代码可能大家未必立马就能理解，但是没关系，我们继续往下看。

```
// 下面的代码其实没啥特别的，就是在设置
// nameserver地址列表,他只不过支持
// 不通过配置的方式来写入nameserver地址，
// 还可以让他发送请求去加载
if (this.brokerConfig.getNamesrvAddr() != null) {

   this.brokerOuterAPI.
       updateNameServerAddressList(
        this.brokerConfig.getNamesrvAddr());

   log.info(
       "Set user specified name server address: {}",
       this.brokerConfig.getNamesrvAddr());

} else if (
    this.brokerConfig.
        isFetchNamesrvAddrByAddressServer()) {

           this.scheduledExecutorService.
               scheduleAtFixedRate(new Runnable() {

              @Override
              public void run() {
                  BrokerController.this.
                      brokerOuterAPI.
                      fetchNameServerAddr();
              }

           }, 1000 * 10, 1000 * 60 * 2, TimeUnit.MILLISECONDS);

    }
```

```

// 下面这段代码其实是处理跟dleger相关的东西的
// 如果你开启了dleger技术，那么其实在下面你会发现会有一些操作
// 但是现在我们都不用过于深究
if (!messageStoreConfig.isEnableDLegerCommitLog()) {

   if (BrokerRole.SLAVE == this.messageStoreConfig.getBrokerRole()) {

       if (this.messageStoreConfig.getHaMasterAddress() != null && this.messageStoreConfig.getHaMasterAddress().length() >= 6) {

           this.messageStore.updateHaMasterAddress(this.messageStoreConfig.getHaMasterAddress());
           this.updateMasterHAServerAddrPeriodically = false;
       } else {
           this.updateMasterHAServerAddrPeriodically = true;
       }

   } else {
       this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

           @Override
           public void run() {
               try {
                   BrokerController.this.printMasterAndSlaveDiff();
               } catch (Throwable e) {
                   log.error("schedule printMasterAndSlaveDiff error.", e);
               }
           }
       }, 1000 * 10, 1000 * 60, TimeUnit.MILLISECONDS);

   }

}

```

```
// 下面这段代码其实也不用管他，一看就是跟文件相关的
if (TlsSystemConfig.tlsMode != TlsMode.DISABLED) {

    // Register a listener to reload SslContext
    try {

        fileWatchService = new FileWatchService(
            new String[] {
                TlsSystemConfig.tlsServerCertPath,
                TlsSystemConfig.tlsServerKeyPath,
                TlsSystemConfig.tlsServerTrustCertPath
            },

            new FileWatchService.Listener() {
                boolean certChanged, keyChanged = false;

                @Override
                public void onChanged(String path) {

                    if (path.equals(TlsSystemConfig.tlsServerTrustCertPath)) {

                        log.info("The trust certificate changed, reload the ssl context");
                        reloadServerSslContext();
                    }
                    if (path.equals(TlsSystemConfig.tlsServerCertPath)) {
                        certChanged = true;
                    }
                    if (path.equals(TlsSystemConfig.tlsServerKeyPath)) {
                        keyChanged = true;
                    }
                    if (certChanged && keyChanged) {

                        log.info("The certificate and private key changed, reload the ssl context");
                        certChanged = keyChanged = false;
                        reloadServerSslContext();
                    }
                }
                private void reloadServerSslContext() {
                    ((NettyRemotingServer) remotingServer).loadSslContext();
                    ((NettyRemotingServer) fastRemotingServer).loadSslContext();
                }
            });
    } catch (Exception e) {
        log.warn("FileWatchService created error, can't load the certificate dynamically");
    }
}

// 最后三行代码，一看就是跟事务消息有关的，初始化事务相关的东西
// 还有就是跟ACL权限控制有关的，包括RPC钩子
// 这些暂时我们也不用去关注他的
initialTransaction();
initialAcl();
initialRpcHooks();
```

3、今天的一点总结

其实如果一定要我说，今天大家看完这些源码，跟着我的注释来走，一方面是对BrokerController初始化的过程有一个大致的印象，另外一方面其实最核心的，你要知道，BrokerController一旦初始化完成过后，他其实就准备好了Netty服务器，可以用于接收网络请求，然后准备好了处理各种请求的线程池，准备好了各种执行后台定时调度任务的线程池。

这些都准备好之后，明天我们就要来讲解BrokerController的启动了，他的启动，必然会正式完成Netty服务器的启动，他于是可以接收请求了，同时Broker必然会在完成启动的过程中去向NameServer进行注册以及保持心跳的。

只有这样，Producer才能从NameServer上找到你这个Broker，同时发送消息给你。

4、今日源码分析作业

今天就请大家自己去把BrokerController的initialize()方法看一下，把里面的源码流程和逻辑过一下，自己也去理解一下，抓住里面的重点，有什么心得体会，都可以发布在评论区跟大家一起交流。

End

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：《付费用户如何加群》（**购买后可见**）